Nerio: Leader Election and Edict Ordering

Robbert van Renesse, Fred B. Schneider, Johannes Gehrke

Department of Computer Science, Cornell University

1 Introduction

Coordination in a distributed system is facilitated if there is a unique process, the *leader*, to manage the other processes. The leader creates *edicts* and sends them to other processes for execution or forwarding to other processes. The leader may fail, and when this occurs a leader election protocol selects a replacement. That protocol satisfies the following properties:

- *Leader Uniqueness*: At any time, at most one process is *leader*.
- *Edict Validity*: Only leaders can create edicts.
- *Edict Ordering*: Recipients of multiple edicts can determine the real-time order in which the edicts were created.
- *Leader Stability*: If a process is leader then, in the absence of failures and in the presence of timely communication and processing, it remains leader.
- *Eventual Election*: If there is no leader then, in the presence of sufficient timely communication and processing and a bounded number of failures, a leader is elected.
- *Fault Tolerance*: Some number of crash failures are tolerated.
- *Efficiency*: The time and storage, processing, and networking resources required by the protocol are reasonable.

We assume that processes can exhibit failures: a process operates correctly until a failure causes that process to stop taking execution steps. Crashed processes are assumed to maintain the values of their variables although these variables are no longer accessible. The clock at a process is a variable and the exposition is simplified if that clock continues to advance even after the process has failed (and the clock is no longer accessible). Message delivery latencies and processing times are assumed to be unbounded. Message loss and reordering by the network is allowed, and network partitioning is permitted too.

Nerio is a class of leader election protocols that implement these properties, Besides developing this class, we derive refinements for two plausible environments: one assumes bounded drift of clock rate with respect to the rate of real

time; the second assumes bounded differences between clock values on any two processes at the same time.

Nerio protocols are based on granting leases [6] and require that failure scenarios are characterized by quorum systems [10], a combination first found in the leader election protocol of Fetzer and Süßkraut in [4]. But leader election properties (Leader Uniqueness, Leader Stability, and Eventual Election) alone offer little value, since a leader may no longer be the leader by the time it sends a message let alone when such a message is received by another process. Our Nerio protocols, which in addition satisfy Edict Validity and Edict Ordering properties, do provide value in asynchronous environments because edicts sent by leaders can be interpreted in the order of their creation, even if the processes that sent the edicts have ceased being leaders.

Formalizing the Properties

Consider a finite set of processes $P = \{p, ...\}$. Let $isLeader_p(t)$ be the property that, at time t, process p is leader. Formally, Leader Uniqueness is the following:

Leader Uniqueness:

$$\forall p, q \in P, t : (isLeader_p(t) \wedge isLeader_q(t)) \Rightarrow (p = q). \tag{1}$$

A leader can create *edicts* that it sends to other processes. For an edict e, define $e.creator$ to be the process that created e, and $e.created$ to be the real time at which e is created. (Note that even the process itself cannot know this time.) Only leaders can create edicts:

Edict Validity:

$$\forall e : isLeader_{e.creator}(e.created) \tag{2}$$

Edict Ordering means that

- There is a total ordering $\prec$ on edicts.
- For edicts e and e', $e.created < e'.created \Rightarrow e \prec e'$.
- Any recipient of edicts e and e' can ascertain whether $e \prec e'$ or $e' \prec e$ holds.

However, Edict Ordering does not imply that receivers all receive the same set of edicts. Let $Order_p(e_1, e_2)$ mean that process p received edicts e_1 and e_2, and believes that e_1 was created before e_2. Formally, Edict Ordering is the following:

Edict Ordering:

$$\forall p, e_1, e_2 : Order_p(e_1, e_2) \Leftrightarrow e_1.created < e_2.created \tag{3}$$

In order to formalize Leader Stability and Eventual Election formally, we assume that there is a time after which message latencies between correct processes are bounded by a known constant d, and there are no more failures. We call this the Global Stabilization Time (`GST`). We do not know when `GST` is, only that it will happen eventually. Then we can have the following properties:

Leader Stability:

$$\exists GST : \forall t_1, t_2 > \texttt{GST},\ p \in P : (isLeader_p(t_1) \wedge t_1 < t_2) \Rightarrow isLeader_p(t_2) \tag{4}$$

Eventual Election:

$$\exists GST : \exists t > \texttt{GST}, p \in P : isLeader_p(t) \tag{5}$$

In Section 2, we describe the Nerio class of leader election protocols that leverage the properties of quorum systems instead of requiring accurate failure detection. Section 3 describes a protocol in this class; it assumes bounded clock drift. Section 4 describes another protocol that assumes that there is a bound on how much two clocks may differ. We compare the two protocols in Section 5. In Section 6 we show how a process can give up its grants to a lease if so desired. Section 7 shows how Nerio protocols support Edict Validity and Edict Ordering. We show that the protocols satisfy Leader Stability in Section 8, while Section 9 demonstrates that the protocols satisfy Eventual Election. A discussion of various issues follows in Section 10. Section 11 discussion prior work.

2 A Class of Leader Election Protocols

Let $\mathcal{Q}$ be a quorum system on P. That is: $\mathcal{Q}$ is a set of process sets such that

$$\forall Q \in \mathcal{Q} : Q \subseteq P \tag{6}$$

$$\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \neq \emptyset \tag{7}$$

An oft-used quorum system consists of all subsets that are majorities in P, that is, $\forall Q \in \mathcal{Q} \Rightarrow |Q| > |P|/2$.

Each process p has the following state variables (we use upper case characters to denote local variables):

C_p (clock): a monotonically increasing clock at process p;

A_p (assignee): a process, initially p itself;

F_p (finish): a clock value measured on the clock of process p, initially 0;

E_p (expiration): another clock value measured on the clock of process p, initially 0.

If X_p is a local variable at process p, then we write $X_p(t)$ for the value of X_p at real time t.

Assume that $C_p(t)$ is continuous and satisfies the following two conditions, which should hold for the clocks found on real processes:

Monotonicity:

$$\forall t_1, t_2 : t_1 < t_2 \Rightarrow C_p(t_1) < C_p(t_2) \tag{8}$$

Growth:

$$\forall T > 0 : \exists t : C_p(t) \geq T \tag{9}$$

In practice, the hardware clock increases in a stepwise fashion rather than continuously. This is not observable if a clock has a sufficiently high resolution relative to the speed at which processes advance. A process can only sample its clock, so by obtaining a value T the process only learns that between the time that the process requested the sample and the time that it obtained the sample, the value of the clock was T. We assume that the clock advances from one sample to the next. This can be ensured by making the clock a pair consisting of the hardware clock and a counter that is reset each time the hardware clock advances and is incremented each time the clock is sampled. This composite clock is then ordered lexicographically. Because of the asynchronous nature of our system, an arbitrary interval may have elapsed between when the sample sample is taken and when it is returned to the process. Therefore, a process cannot tell the difference between a clock that increases continuously, and one that does not.

Assuming that C_p increases we can define an inverse function $c_p(T)$ on clocks with the following properties:

$$C_p(c_p(T)) = T \tag{10}$$

$$c_p(C_p(t)) = t \tag{11}$$

Lemma 2.1

$$\forall p \in P, t, T : C_p(t) < T \Leftrightarrow t < c_p(T)$$

Let $\gamma_{p,q}(t)$ be the predicate

$$\gamma_{p,q}(t) \quad \equiv \quad A_q(t) = p \ \wedge \ C_q(t) < F_q(t).$$

If $\gamma_{p,q}(t)$ holds, we say that, at time t, *process q grants a lease to process p*. Note that a process cannot grant a lease to two different processes at the same time t, because a variable (*e.g.*, $A_q(t)$) can have only one value at time t.

We can now define formally what it means to be leader:

$$isLeader_p(t) \equiv \exists Q \in \Omega : (\forall q \in Q : \gamma_{p,q}(t)) \tag{12}$$

That is, process p is leader at time t iff a quorum of processes grant a lease to p at time t. It should be clear to the reader that (12) implies (1): because of the

intersection property of quorums (Equation (7)) there cannot be two different quorums, one in which all processes are granting a lease to p_1, and another quorum in which all processes are granting a lease to a different process p_2, at the same time.

We need an implementation of $isLeader_p$. Each process p has a variable E_p, which gives an expiration time of p's leadership, initially 0. Like F_p, E_p is measured on p's clock. In Nerio protocols, the following invariant holds:

$$\forall p, t : (C_p(t) < E_p(t)) \Rightarrow isLeader_p(t) \tag{13}$$

(The implication holds only in one direction because, as we shall see, processes extend their grants conservatively, and thus it may be that a quorum of processes are still granting a lease to p after p gives up on the lease.)

Combining (13) and (12) and substituting $\gamma_{p,q}(t)$, we get the following property:

$$\forall p, t : (C_p(t) < E_p(t)) \Rightarrow (\exists Q \in \mathcal{Q} : (\forall q \in Q : A_q(t) = p \wedge C_q(t) < F_q(t))) \tag{14}$$

We take this as the defining characteristic of a Nerio class leader election protocol.

At this point it is useful to consider what happens if a process crashes. By the *Growth* condition (Equation (9)), the clock of the process continues increasing. We need this in order to ensure that if a crashed process p was a leader, eventually it stops being leader (because $C_p(t) < E_p(t)$ becomes false), and if a crashed process q granted a lease, eventually this lease expires (because $C_q(t) < F_q(t)$ becomes false). Since a crashed process cannot produce any output, having the clock stop is indistinguishable from a clock that continues to increase.[1]

Below we will show examples of protocols that maintain (14), given certain assumptions about the environment.

3 Clocks with Bounded Drift

Assume the drift (accuracy of rate) of each clock is bounded by a constant ρ per time unit. That is:

$$\forall p \in P, t, \delta : C_p(t) + (1 - \rho)\delta \leq C_p(t + \delta) \leq C_p(t) + (1 + \rho)\delta. \tag{15}$$

In other words, during a real-time period δ, the clock of a process may advance by as little as $(1 - \rho)\delta$, or as much as $(1 + \rho)\delta$.

In the Nerio class protocol that we derive in this section, a process p never decreases F_p. Thus the protocol maintains the following invariant:

$$\forall p \in P, t_1, t_2 : t_1 < t_2 \Rightarrow F_p(t_1) \leq F_p(t_2). \tag{16}$$

[1] In practice, a hardware clock often continues to increase for some amount of time as it is backed up by an internal battery.

Furthermore, consistent with the meaning of a lease, a process p never changes A_p if $C_p < F_p$. As a result, once a process p has granted a lease to A_p, this grant remains until real time $c_p(F_p)$.

A process p trying to become leader (or extend the period during which it is leader) executes the following algorithm, which we call `Obtain Quorum with Bounded Drift`, or `OQwBD` for short. Process p uses a temporary $Start_p$ into which it stores the starting time of the algorithm:

1. set $Start_p := C_p$ (sample starting time);
2. select a real time period δ, $\delta > 0$;
3. broadcast $\langle \texttt{grantRequest}, p, Start_p, \delta \rangle$.

Upon receipt of a `grantRequest` message, a process q does the following:

4. $T_q := C_q$ (save local time into a temporary variable T_q);
5. if $p \neq A_q \wedge T_q < F_q$, then ignore the request (q is already granting a lease to A_q, $A_q \neq p$);
6. otherwise
   - 6.1. $A_q := p$; $F_q := \max(F_q, T_q + (1+\rho) \cdot \delta)$;
   - 6.2. send $\langle \texttt{ok}, q, Start_p \rangle$ to p.

Meanwhile, process p waits for `ok` messages:

7. wait for a $\langle \texttt{ok}, q, Start_p \rangle$ from each process q in a quorum of $\mathcal{Q}$ or until $C_p \geq Start_p + (1-\rho) \cdot \delta$;
8. if `ok` messages are received from a quorum and $C_p < Start_p + (1-\rho) \cdot \delta$, then $E_p := Start_p + (1-\rho) \cdot \delta$ (we say that the `OQwBD` algorithm *completed*);
9. if not a sufficient number of `ok` responses are received by $C_p \geq Start_p + (1-\rho) \cdot \delta$, then this instantiation of `OQwBD` *failed*.

By measuring $(1-\rho) \cdot \delta$ on its local clock, process p will stop believing it is leader before *at most* δ real time units have expired since p initiated `OQwBD`. A process q, by measuring $(1+\rho) \cdot \delta$, grants the lease for *at least* δ real time units since process p started `OQwBD`. (Process q calculates a maximum in order to ensure that F_q can only progress forwards as required by (16)).

Process p can run `OQwBD` at any time. We say that p *aborts* `OQwBD` if p starts a new execution before the current one completed. Once aborted, responses for the earlier instantiation of `OQwBD` will be ignored. The clock value $Start_p$ is included in the messages only to identify an instantiation of `OQwBD` (the tuple $(p, Start_p)$ uniquely identifies an instantiation of `OQwBD`); receivers do not interpret the clock value, but return it in the response. This way, process p can ignore responses of aborted instantiations.

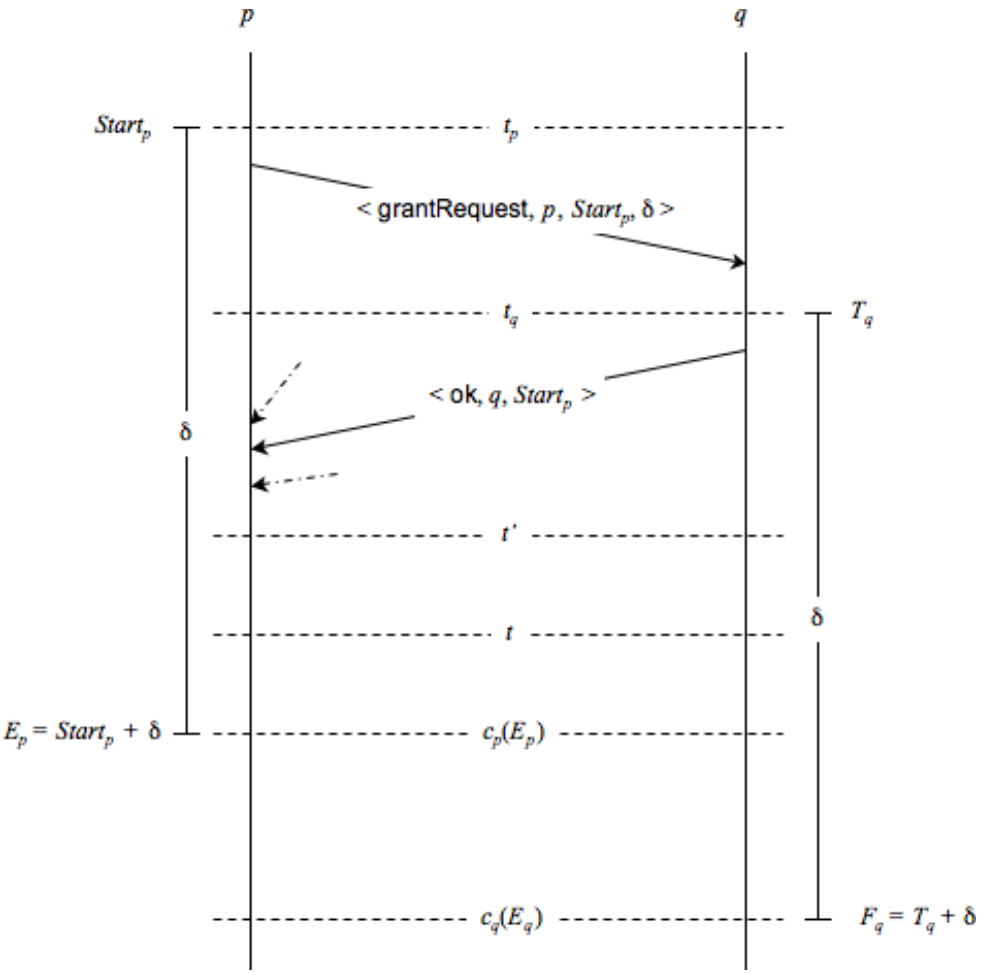

Figure 1: Example of a protocol exchange between a process p that initiated the protocol and a process q that granted the lease and is included in the quorum that p uses to complete the protocol. For clarity, the drift $\rho = 0$. Dashed horizontal lines indicate real time. The labels to the left are clock values on p's clock; the labels on the right are clock values on q's clock.

To prove that (14) holds for any p and t, consider a process p and a time t for which $C_p(t) < E_p(t)$ holds, and note that $t < c_p(E_p(t))$ (Lemma 2.1). Let t' be the time at which the last instantiation of `OQwBD` completed at process p. Note that p updates E_p only at this time, and since this is the last instantiation of the protocol, we have the following:

$$E_p(t) = E_p(t') \tag{17}$$

Let t_p be the time at which p assigned $Start_p$ in this last instantiation of `OQwBD`, and thus

$$Start_p(t') = Start_p(t_p) \tag{18}$$

We have to show that there exists a quorum Q so that $\forall q \in Q : A_q(t) = p \ \wedge \ C_q(t) < F_q(t)$. We show that the quorum that responded to p and caused p to complete `OQwBD` is such a quorum.

Let Q be the quorum that responded to p. Consider a process $q \in Q$ and let t_q be the time at which some process q sampled the local clock resulting in a value T_q, so that $T_q = C_q(t_q)$. Note that

$$t_p \leq t_q \leq t' \leq t < c_p(E_p(t)). \tag{19}$$

(See Figure 1 for an illustration in the case $\rho = 0$.)

Lemma 3.1 $c_p(E_p(t)) \leq c_q(F_q(t))$.

Proof

(1)	$E_p(t) = E_p(t')$	(Equation (17))
(2)	$E_p(t') = Start_p(t') + (1-\rho)\delta$	(Algorithm `OQwBD`)
(3)	$Start_p(t') = Start_p(t_p)$	(Equation (18))
(4)	$Start_p(t_p) = C_p(t_p)$	(Algorithm `OQwBD`)
(5)	$C_p(t_p) + (1-\rho)\delta \leq C_p(t_p + \delta)$	(Equation (15))
(6)	$E_p(t) \leq C_p(t_p + \delta)$	(Combining (2) thru (5))
(7)	$c_p(E_p(t)) \leq t_p + \delta$	(Lemma 2.1)
(8)	$F_q(t) \geq F_q(t_q)$	(Equations (19) and (16))
(9)	$F_q(t_q) = C_q(t_q) + (1+\rho)\delta$	(Algorithm `OQwBD`)
(10)	$C_q(t_q) + (1+\rho)\delta \geq C_q(t_q + \delta)$	(Equation (15))
(11)	$F_q(t) \geq C_q(t_q + \delta)$	(Combining (8), (9), (10))
(12)	$t_q + \delta \leq c_q(F_q(t))$	(Lemma 2.1)
(13)	$(t_p + \delta) \leq (t_q + \delta)$	(Equation (19))
(14)	$c_p(E_p(t)) \leq (t_p + \delta) \leq (t_q + \delta) \leq c_q(F_q(t))$	(Combining (7), (12), (13))

∎

It remains to show that $A_q(t) = p \wedge C_q(t) < F_q(t)$. This follows directly from $t_q \leq t' \leq t < c_p(E_p(t)) \leq c_q(F_q(t))$. After assigning A_q and F_q between t_q and t', A_q cannot be changed until $c_q(F_q(t))$ at the earliest.

Note that no effort is made to detect process crashes. If $isLeader_p(t)$ at the time a process p crashes, that process continues to be leader until there is no longer a quorum of processes that grant a lease to p.

4 Clocks with Bounded Skew

Our second instance of a Nerio class leader election protocol is similar to the first, but instead of assuming bounded drift (Equation (15)) we assume that clocks at any two processes always differ by at most Δ:

$$\forall p, q, t : -\Delta \leq C_p(t) - C_q(t) \leq \Delta. \tag{20}$$

We present a new algorithm called `Obtain Quorum with Bounded Skew`, or `OQwBS` for short. (Skew is the difference between two clock values at the same time.) The variables of `OQwBS` are the same as those of `OQwBD`. A process p can initiate `OQwBS` as follows:

1. set $Start_p := C_p$ (sample starting time);
2. select a time period δ, $\delta > 0$;
3. broadcast $\langle \texttt{grantRequest}, p, Start_p, \delta \rangle$.

Upon receipt, a process q does the following:

4. $T_q := C_q$ (sample local time);
5. if $p \neq A_q \wedge T_q < F_q$, then ignore the request;
6. otherwise
   - 6.1. $A_q := p$; $F_q := \max(F_q, Start_p + \delta + \Delta)$;
   - 6.2. send $\langle \texttt{ok}, q, Start_p \rangle$ to p.

Note that because any two clocks differ by at most Δ, q can interpret $Start_p$ with respect to its own clock. As before, process p waits for `ok` responses:

7. wait for a $\langle \texttt{ok}, q, Start_p \rangle$ from each process q in a quorum of Ω or until $C_p \geq Start_p + \delta$;
8. if `ok` messages are received from a quorum and $C_p < Start_p + \delta$, then $E_p := Start_p + \delta$ (we say that the `OQwBS` algorithm *completed*);
9. if not a sufficient number of `ok` responses are received by $C_p \geq Start_p + \delta$, then this instantiation of `OQwBS` *failed*.

Again, the proof of Leader Uniqueness is based on showing that $c_p(E_p(t)) \leq c_q(F_q(t))$, and it is easy to see why this is true.

5 Comparison

In the `OQwBD` protocol of Section 3, a process q may grant a lease for a process p long beyond $c_p(E_p(t))$ if the `grantRequest` message to p is delayed that much. If p has failed, this grant could prevent other processes from becoming leader. It thus appears that the `OQwBS` protocol of Section 4, which is based on bounded skew, has an important advantage. However, below we will argue that in practice bounded drift is more likely to be guaranteed than bounded skew, so `OQwBD` is likely to be more robust in practice.

Hardware clock manufacturers often specify a bound on clock drift, and this bound is typically within the range of 10^{-7} to 10^{-5} given a sufficiently stable temperature within the casing of a computer chassis. For performance measurements, in which it is necessary to measure the passage of time, rather than to tell what time it is, operating systems usually provide access to the raw clock value, as opposed to one that may be adjusted by a clock synchronization protocol attempting to reduce skew.

Under virtualization, the hardware clock may be virtualized, and drift would no longer be bounded. Fortunately, Xen allows guests to sample the hardware clock. Under VMware, the hardware clock is not directly accessible. Fortunately, VMware does make CPU performance counters accessible, including a way to measure the passage of time. If this facility documents a bound on drift, then this is enough for our purposes. However, if a virtual machine is migrated, a clock may jump arbitrarily, violating the assumptions that we make on clocks.

The protocol based on bounded skew allows processes to leverage a bound Δ to avoid a process granting a lease more than Δ beyond $c_p(E_p(t))$. Bounded skew requires a *clock synchronization algorithm*. Clock synchronization algorithms require bounded latency on communication and bounded execution times, in addition to requiring bounded clock drift. In the absence of such bounds, clock synchronization algorithms such as NTP provide, at best, probabilistic bounds on skew (with unspecified probability).

Below we will only assume bounded clock drift and not bounded skew, although the results are generalized easily.

6 Releasing Grants

It is sometimes useful for a process to give up the grants it received. For example, if a process is not able to obtain grants from a quorum, and thus does not have a lease on leadership, then it might as well give up the grants that it has so that perhaps another process can be more lucky. Even if a process did obtain a lease and became leader, it may for some reason give up its leadership by releasing its grants. In this section we will show how this can be done without violating invariant 14.

A process p that wants to release its grants first aborts any instance of `OQwBD` that it may be running. Second, process p sets $E_p(t)$ to $C_p(t)$. We note that it is always safe for a process p to do so as this cannot affect the validity of

invariant 14. If p was leader, it will no longer be leader as a result. So at this point, p is neither leader nor is it trying to become one.

Next p broadcast a request to all peers to release its grant. A process q that receives such a message from p will check to see if $p = A_q$. If not, it ignores the request. If so, it will set F_q to C_q, causing the grant to expire immediately, *even if* $F_q > C_q$. The reader will notice that this violates invariant 16, which was used in Lemma 3.1 to proof that $c_p(E_p(t)) \leq c_q(F_q(t))$. However, this lemma was only shown to hold when $C_p(t) < E_p(t)$, and because p has reset E_p to C_p, this precondition no longer holds. Invariant 16 is not used elsewhere, and thus expiring the grant does not violate invariant 14.

7 Edicts

Leaders create edicts, which they send to one or more processes. Because of a lack of assumptions about message latencies and process execution speeds, such an edict may take an arbitrary amount of time to arrive at a process, and may even be lost in the network. Edicts may also be stored or forwarded. So an old edict may be delivered after an edict that was created more recently, possibly by a different leader. Edict Ordering prevents chaos: it ensures that any two different edicts can be compared and ordered in a manner consistent with the real times of their creation.

For this ordering to make sense, the time an edict is created must be defined properly. The last step by a process p creating an edict e is to sample C_p, obtaining a value $T = C_p(t)$. In order to ensure Edict Validity, the process determines if $T < E_p$. If so, then t is the creation time of edict e, that is, $e.created = t$. (Unfortunately, even the leader itself cannot determine t.) If not, then the edict creation fails, because at time t, process p may not have been leader.

We describe how $Order_p(e_1, e_2)$ in Equation (3) is implemented for algorithm `OQwBD`, but the idea does not depend on specifics of `OQwBD`. Extend the `ok` response (Step 2) from q with T_q, such that q sends $\langle \texttt{ok}, q, T_q, Start_p \rangle$ to p. Process p awaits messages from all processes in a quorum $Q \in \mathcal{Q}$, and constructs a *Quorum Timestamp* QT_p as the set of pairs (q, T_q) for all $q \in Q$. In addition, process p maintains an *Edict Counter* EC_p, initially 0, counting the number of edicts created by p.

Every time p creates an edict, it tags that edict with an *Edict Timestamp* (QT_p, EC_p) and increments EC_p. Before sending the edict, process p checks to see if $C_p < E_p$ to make sure it is still leader. If not, the edict is not valid and should be discarded.

We define an ordering on Edict Timestamps and show it consistent with the real time in which the edicts were created. Edict timestamps are lexicographically ordered, first by quorum timestamp and then by the natural ordering on edict counters. Quorum timestamps are ordered as follows:

$$QT_1 < QT_2 \Leftrightarrow (\exists q, T_1, T_2 : (q, T_1) \in QT_1 \wedge (q, T_2) \in QT_2 \wedge T_1 < T_2) \quad (21)$$

We show that this ordering is consistent with the creation times of edicts. Let X be a completed instantiation of `OQwBD`. X has the following attributes:

$X.owner$	the process that initiated X and became leader
$X.start$	the real-time when X started (*i.e.*, $c_{X.owner}(Start_{X.owner})$)
$X.completion$	the real-time when X completed
$X.QT$	the quorum timestamp that $X.owner$ generated
$X.expiration$	the real-time when X expires (*i.e.*, $c_{X.owner}(E_{X.owner})$)

Some trivial observation about such an X are:

$$X.start \leq X.completion < X.expiration \tag{22}$$

$$\forall t : (X.completion \leq t < X.expiration) \Rightarrow isLeader_{X.owner}(t) \tag{23}$$

$$\forall q, T : (q, T) \in X.QT \Rightarrow X.start \leq c_q(T) \leq X.completion \tag{24}$$

We order instantiations by their completion time, that is, $X < X' \Leftrightarrow X.completion < X'.completion$.

Lemma 7.1 $\forall X, X' : X < X' \Rightarrow X.QT < X'.QT$.

Proof By contradiction, assume there can exist an X and X' such that $X < X'$ (and thus $X.completion < X'.completion$) and $\neg(X.QT < X'.QT)$. Because quorums overlap, there must exists a q, T, T' such that $(q, T) \in X.QT$ and $(q, T') \in X'.QT$. By assumption, $T \geq T'$ (for otherwise $X.QT < X'.QT$). We consider two cases.

- $X.owner = X'.owner$: Then it must be the case that $X.completion < X'.start$ (or X would have been aborted and could not have completed). From (24) it must be that $T < T'$, contradicting the assumption that $T \geq T'$.
- $X.owner \neq X'.owner$: From time $c_q(T)$ until $X.completion$ (and beyond), q has granted a lease to $X.owner$, and similarly, from $c_q(T')$ to $X'.completion$, q has granted a lease to $X'.owner$. Because $X.completion < X'.completion$ and $c_q(T) \geq c_q(T')$, it must be the case that at time $X.completion$, process q has granted a lease both to $X.owner$ and $X'.owner$. But a process cannot grant leases to two different processes at the same time. ∎

Note, as a corollary, that quorum timestamps are well-ordered, consistent with the ordering on instantiations of `OQwBD`.

8 Leader Stability

When there is a leader, Leader Stability implies that the leader persists in that role in the absence of failures and while messages are delivered and processed in

a timely fashion. Suppose that message round-trip time is bounded by a known constant d. In that case, if leader p starts `OQwBD` before $c_p(E_p) - d$, then it is able to extend its leadership before it expires. So p should use $\delta > 2d$ in order that in the next period of leadership it is able to do so again. Choices of δ are discussed in Section 10.1.

9 Eventual Election

If there is no leader (*i.e.*, $\forall p : C_p \geq E_p$) then multiple processes could try to become leader. There is no guarantee that any will succeed, however. But if we could somehow ensure that only one process p executes `OQwBD`, and the process waits long enough to do so (so that all F_*'s have expired), then `OQwBD` is guaranteed to succeed eventually (after `GST`). This would seem to create a circularity, as as choosing p requires solving leader election. The way out is to use a weak version of leader election (which may select multiple *weak leaders*) in order to make successful completion of `OQwBD` likely. The more likely it is that weak leader election selects only a single weak leader, the more likely an instantiation of `OQwBD` terminates successfully. In addition, for Eventual Election to hold, after `GST` the weak leader election protocol is required to produce a single weak leader.

Here is such a weak leader election algorithm: Assume processes in P are ordered, that is, $p < q < ...$, and elect the smallest process in P that has not failed. To this end, processes are organized into a virtual ring in order. The scheme uses a failure detection algorithm such as simple pinging or the more sophisticated ϕ-accrual failure detector [7] that gives a better approximation of the failure status of processes. Each process pairs with the closest predecessor and closest successor on the ring that it considers correct by the failure detector, and monitors it. If a process q believes it is the lowest correct process (because the identifier of its predecessor is larger than its own), then it considers itself a weak leader. Note that under the properties of `GST`, failure detection becomes accurate and the algorithm will produce a single leader.

A weak leader initiates `OQwBD` to try to become a leader if $A_q \neq q \wedge C_q < F_q$ (*i.e.*, it is not currently granting a lease to another process). It does so periodically in order to deal with possible collisions and message loss.

The Eventual Election property states that under the conditions that hold after `GST`, a leader will eventually be chosen by the Nerio protocol if there is none yet (and, because of Leader Stability, it will remain leader henceforth). To see why Eventual Election holds for the presented protocols, note that eventually only one process will attempt to become leader because of the properties of weak leader election. After all conflicting grants have expired, and because round-trip latencies are bounded by d, eventually this process will complete `OQwBD`.

10 Discussion

10.1 Choice of δ

A process that initiates `OQwBD` chooses some value for δ. No matter what value of δ is chosen, Leader Uniqueness will hold, but choosing δ too small could adversely affect Leader Stability and Eventual Election. Therefore δ should be chosen large enough so a leader can extend the period of its leadership without interruption, but short enough so that recovery can be swift after the leader fails.

Suppose d is an estimate for the round-trip time, and represents that, say, in 99.9% of round-trips the round-trip latency is less than d. A leader might initiate `OQwBD` to extend its lease before $E_p - (1+\rho) \cdot d$ measured on its local clock. Clearly, δ should be chosen larger than d plus the time that remains on the lease, which can be conservatively estimated by p as $(1+\rho) \cdot (E_p - C_p)$.

In practice, d is likely no larger than a few milliseconds on today's hardware, assuming processing of Nerio messages receive a high priority, and ρ is likely no more than 10 microseconds. But choosing δ as small as possible would likely result in too many round-trips per second. If we want space out instantiations of `OQwBD` by at least i time units, then we should choose $\delta = d + \max(i, (1+\rho) \cdot (E_p - C_p))$.

10.2 Interference

If more than one process concurrently tries to become leader, then none may be able to enlist a quorum. They each would then have to wait to let conflicting grants expire before attempting to rerun `OQwBD`.

In both proposed Nerio protocols, processes respond to the initiator only if they grant the lease request. But there is something to gain if the protocols are modified so that if a process has granted a lease to another process, then instead of just ignoring the grant request, it responds with an error message. The error message helps an initiator to determine if there is hope of obtaining a quorum.

The protocols can be extended with revocation requests to further avoid interference. This further extension requires that grant and revocation requests from the same source are delivered in FIFO order. When a process q receives a revocation request from a process p, and if $A_q = p \wedge C_q < F_q$, then q sets F_q to C_q, thereby releasing its grant to p. (The FIFO order ensures that delayed revocation request do not inadvertently revoke outstanding grants.) Obviously, a process p that sends a revocation request must first have aborted the protocol, thus even if it ends up collecting positive responses from a quorum, E_p should not be advanced.

10.3 Network Partitioning

Nerio class protocols work work even if the network partitions if there is a partition that contains a quorum of correct processes. And if there is no such partition or if functionality is desired in minority partitions (*i.e.*, partitions that do not hold a quorum of processes), then the weak leader election algorithm might be used to assign a temporary, non-authoritative, leader in each partition that can provide partial functionality.

10.4 Finding the Leader

What if an external client, seeking that an edict be issued, sends a request to a process p in P but p is not currently leader? If process p has an unexpired grant for another process q, then process p can respond by giving q as a forwarding address. If not, process p may attempt to become leader. Failing that, p may buffer the request until a leader emerges, or return an error response.

10.5 Leader Verification

A process q may want to check whether some other process p is leader. The following protocol, based on bounded drift, will accomplish this:

1. set $\mathit{Start}_q := C_q$ (save starting time);
2. send $\langle \texttt{verifyLeadership}, q, \mathit{Start}_q \rangle$ to p.

Upon receipt of a `verifyLeadership` message, a process p does the following:

3. calculate $\delta := (E_p - C_p)/(\rho + 1)$;
4. send $\langle \texttt{remainder}, p, \delta, \mathit{Start}_q \rangle$ to q.

Here δ equals the minimal amount of real time that is left of p's leadership. Note that if p is no longer leader, δ will be negative. If q receives the response, it calculates $T = \mathit{Start}_q + \delta \cdot (\rho - 1)$, and as long as $T < C_q$ holds, p is guaranteed to be leader (and possibly a bit longer than that depending on rate differences between C_p and C_q).

10.6 Changing Membership

Nerio protocols can be adapted to handle the case where P changes over time. We introduce *epochs*, numbered consecutively starting at 0. Each epoch e is associated with a set of processes P_e and quorum system $\mathcal{Q}_e$ defined on P_e. For simplicity, assume different epochs have non-overlapping sets of processes:

$$\forall e, e' : e \neq e' \Rightarrow P_e \cap P_{e'} = \emptyset. \tag{25}$$

(In practice, a process that is a member of more than one epoch should maintain different copies of its state variables for each epoch.)

Each epoch is defined to be `PENDING`, `RUNNING`, or `TERMINATED`. Each epoch starts in the `PENDING` state, except for epoch 0 which starts in the `RUNNING` state. At any point in time, at most one epoch is in the `RUNNING` state, and all prior epochs are `TERMINATED`.

If epoch e is `RUNNING`, it can be terminated by getting each process q in some quorum of $\mathcal{Q}_e$ to set $A_q = \perp \wedge F_q = \infty$. (A process can only do so if there is no current outstanding grant.) We say that process q is *wedged* if $A_q = \perp \wedge F_q = \infty$ holds. Once a quorum of processes are wedged, no process can become leader in that epoch. At this same time, epoch $e+1$ automatically becomes `RUNNING`. That is, an epoch is defined to be `RUNNING` iff all prior epochs are `TERMINATED` and no quorum of processes are all wedged in that epoch.

A process p in epoch $e+1$ ignores grant requests, and does not send any, until it has learned that epoch e is `TERMINATED`. Process p can learn that e is `TERMINATED` by querying processes in a quorum of $\mathcal{Q}_e$ and detecting that these are wedged, or by receiving a grant request from a process in P_{e+1}.

Note that $isLeader_p(t)$ will hold only if epoch e is `RUNNING` at time t and $p \in P_e$ holds. Because at most one epoch is `RUNNING`, Leader Uniqueness continues to hold, even given multiple epochs.

Leader Stability no longer makes sense because epoch memberships are non-overlapping. However, an epoch $e+1$ that wants to start running could have a particular process $p \in P_{e+1}$ be in charge of wedging the processes in P_e, by sending a $\langle \texttt{grantRequest}, \perp, Start_p, \infty \rangle$ message to these processes, and upon obtaining `ok` responses from a quorum of those processes, send a regular grant request to the processes of epoch P_{e+1}. Thus the new epoch has significant control over which process it wants to be leader initially.

Once a process receives a grant request with $\delta = \infty$, but has an outstanding (normal) grant request, it could buffer the special grant request and grant it upon expiry of the current lease. In that case, if at most a quorum of processes in a `RUNNING` epoch are faulty, eventually the epoch will become `TERMINATED` and the processes of the next epoch will be able to learn so. Therefore Eventual Election also continues to hold.

The reconfiguration protocol should be invoked when processes are suspected of having crashed, or eventually there may no longer be a quorum available to elect a leader. The reconfiguration protocol can only make progress if a quorum in Q_e is correct and can be wedged, and thus if too many processes crash, it is no longer possible to reconfigure. Under manual intervention, an administrator could explicitly mark certain processes as having failed. The quorum system could then be adjusted with smaller quorums in order to make progress.

Note that edict timestamps can be extended with epochs in order to make sure the Edict Ordering continues to hold.

11 Related Work

Leader election is used in practical systems. For example, the IEEE 1394 "Firewire" serial bus standard, for the purpose of coordination among devices,

includes such a protocol that creates a spanning tree of devices with a unique root acting as leader. Early work on leader election focused on efficiently finding extremas (the node with the minimum or maximum identifier) in a connected network topology of unknown size. The problem was apparently first formulated and solved in 1977 by Gerard LeLann [8]. Many papers on this subject have appeared since.

In 1982, Hector Garcia-Molina defined the problem of leader election in a distributed system that admits failures [5], and presented protocols. That paper includes separate definitions for synchronous and asynchronous systems. For a synchronous system, Garcia-Molina's definition of leader election requires that there be at most one leader at at time, and in the absence of failures a leader is elected within a fixed time limit. For an asynchronous system, the definition applies only to those nodes that experience synchronous communication—the other nodes may end up with different leaders.

Consensus protocols [2] can be used to solve leader election in both synchronous and asynchronous systems. Each participant proposes itself as leader, and the consensus protocol subsequently decides on one of the proposals. Dividing time into time slots, an instantiation of consensus could be used for each time slot. Doing so would lead to unnecessarily high overhead, and many consensus protocols rely on leader election themselves, creating a circularity.

Fueled by leader-based consensus protocols, many papers discuss leader election in partially asynchronous systems. In this formulation, a protocol may output multiple leaders, but there must exist a time after which the protocol output exactly one leader. In asynchronous environments these protocols are probabilistic, producing a single leader in case the environment is reasonably timely, but that may produce multiple leaders in case the environment is not. We call this *weak leader election*, but it is also referred to as *local leader election*.

Weak leader election in asynchronous environments is closely related to the failure detection problem, whereby a leader is the node with the lowest (or highest) identifier that is not suspected of having failed. Fetzer and Cristian [3] study the problem of weak leader election in partitionable networks, and use a technique based on leases [6] (to define partition boundaries). Stable (but weak) leader election was considered in [1]. A performance comparison of three recent stable leader election algorithms appears in [9]. This paper also consider dynamic membership.

The problem of strong leader election in a partially synchronous environment was discussed by Fetzer and Süßkraut in [4]. Their protocol uses leases and quorums. The Nerio protocols described in this paper generalize this idea by defining an invariant (Equation (14)) that all such protocols must satisfy, and can be used to transform any weak leader election protocol into one that is both strong and stable, and support dynamic membership.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proc. of the 15th International Symposium on DIStributed Computing*, pages 108–122, Lisbon, Portugal, October 2001. Springer-Verlag.

[2] M. Barborak and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2), 1993.

[3] C. Fetzer and F. Cristian. A highly available local leader election service. *IEEE Transactions on Software Engineering*, 25(5):603–618, 1999.

[4] C. Fetzer and M. Süßkraut. Leader Election in the Timed Finite Average Response Time Model. In *Proceeddings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 375–376, 2006.

[5] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59, January 1982.

[6] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the Twelfth ACM Symp. on Operating Systems Principles*, pages 202–210, Litchfield Park, AZ, November 1989.

[7] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 66–78, October 2004.

[8] G. LeLann. Distributed systems–towards a formal approach. *Information Processing*, 77:155–160, 1977.

[9] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *Proc. of the Int. Conf. on Dependable Systems and Networks DSN O8*, pages 207–216, Anchorage, AK, June 2008.

[10] R.H. Thomas. A solution to the concurrency control problem for multiple copy data bases. *Proc. of COMPCON'78*, pages 88–93, 1978.